

SQL Cheatsheet

github.com/reidenong/cheatsheets, for general usage.

Data Types (PostgreSQL)

BOOLEAN	TRUE / FALSE
INTEGER	signed 4-byte integer
DOUBLE PRECISION	IEEE-754 8-byte floating-point
CHAR(n)	fixed-length character string (pads with spaces)
VARCHAR(n)	variable-length character string (max n, no pad)
TEXT	variable-length with no limit
DATE	calendar date (year-month-day)
TIMESTAMP	date and time (no time zone)

Creating tables

Employees(id: integer, name: text),

```
CREATE TABLE Employees (
    id INTEGER,
    name VARCHAR(50)
);
```

```
CREATE TABLE IF NOT EXISTS book (
    title VARCHAR(256) NOT NULL,
    ISBN13 CHAR(14) PRIMARY KEY
);
```

Default values

```
CREATE TABLE Employees (
    id INTEGER,
    name VARCHAR(50) DEFAULT 'John'
);
```

Inserting Data

Specifying all attribute values

```
INSERT INTO Employees VALUES
(101, 'Sarah', 25, 'dev');
```

Specifying selected attribute values

```
INSERT INTO Employees (id, name) VALUES
(101, 'Sarah');
```

Non specified attributes are null.

Deleting Data

```
DELETE FROM Employees
WHERE id = 1;
```

Updating Data

Updating specific data

```
UPDATE Employees
SET age = age + 1
WHERE name = 'Alice';
```

Updating entire column

```
UPDATE Employees
SET age = 0;
```

Handling NULL

- Null checks: x IS NULL, x IS NOT NULL
- Distinct: x IS DISTINCT FROM y (inequality that also handles NULL)

Rounding

```
SELECT ROUND(123.45678, 2); -- 123.46
```

Date Manipulation

Current Datetime

```
SELECT CURRENT_DATE;
SELECT CURRENT_TIMESTAMP;
```

Date Literal

```
DATE '2021-02-01' -- YYYY-MM-DD
```

Date Increments

```
SELECT CURRENT_DATE + INTERVAL '7 day'; -- 7
days ahead
```

Date Truncation

```
SELECT DATE_TRUNC('month', your_timestamp);
-- '2025-10-01 00:00:00'
SELECT TO_CHAR(your_timestamp, 'YYYY-MM');
-- '2025-10'
```

Data Integrity Constraints

Primary Key

```
CREATE TABLE Employees (
    id INTEGER PRIMARY KEY,
    id INTEGER UNIQUE NOT NULL --alt
);
```

Not Null constraint

```
CREATE TABLE Employees (
    name VARCHAR(50) NOT NULL,
    name VARCHAR(50)
    CONSTRAINT nn_name NOT NULL --alt
);
```

Unique Constraint

```
CREATE TABLE Employees (
    id INTEGER UNIQUE,
    id INTEGER
    CONSTRAINT uq_id UNIQUE --alt
);
```

Checks

```
CREATE TABLE Employees (
    age INTEGER CHECK (age > 0),
    role VARCHAR(50) NOT NULL
);
```

Table Constraint

```
CREATE TABLE Employees (
    id INTEGER,
    name VARCHAR(50),
    UNIQUE(id, name),
    CONSTRAINT u_alloc UNIQUE(id, name) --alt
);
```

Foreign Key Constraints

Each foreign key in referencing relation must

- appear as primary key in referenced relation OR
- be a null value

```
CREATE TABLE Teams (
    tid INTEGER PRIMARY KEY,
    FOREIGN KEY (eid) REFERENCES Employees(id),
    CONSTRAINT fkey FOREIGN KEY (eid)
        REFERENCES Employees(id) -- alt
);
```

Foreign Key Violations

```
FOREIGN KEY (eid) REFERENCES Employees(id)
ON DELETE <Action>
ON UPDATE <Action>
```

- NO ACTION : Reject delete/update if it violates constraint
- RESTRICT : NO ACTION but check cannot be deferred
- CASCADE : Propagates delete/ update to referencing tuples
- SET DEFAULT : Updates foreign keys to some default value
- SET NULL : Updates foreign keys to null if column allows

Deferrable constraints

```
CONSTRAINT valid_lifetime
    CHECK (start_year <= end_year)
    DEFERRABLE INITIALLY DEFERRED,
CONSTRAINT valid_lifetime
    CHECK (start_year <= end_year)
    DEFERRABLE INITIALLY IMMEDIATE
);
```

- INITIALLY IMMEDIATE : Validates immediately, but can be temporarily avoided; strict by default, flexible if needed
- INITIALLY DEFERRED : Automatically delays till end of transaction to validate. relaxed by default

Modifying a Database with SQL

Change Data Type

```
ALTER TABLE Projects
ALTER COLUMN name TYPE VARCHAR(200);
```

Set Default Value

```
ALTER TABLE Projects
ALTER COLUMN start_year SET DEFAULT 2020;
```

Drop Default Value

```
ALTER TABLE Projects
ALTER COLUMN start_year DROP DEFAULT;
```

Add Column

```
ALTER TABLE Projects
ADD COLUMN budget NUMERIC DEFAULT 0.0;
```

Drop Column

```
ALTER TABLE Projects
DROP COLUMN budget;
```

Add Constraint

```
ALTER TABLE Teams
ADD CONSTRAINT eid_fkey FOREIGN KEY (eid)
REFERENCES Employees (id);
```

Drop Constraint

```
ALTER TABLE Teams
DROP CONSTRAINT eid_fkey;
```

Drop Table

```
DROP TABLE Projects;
```

```
DROP TABLE IF EXISTS Projects;
```

```
DROP TABLE Projects CASCADE;
```

Querying

```
SELECT [DISTINCT] target-list
FROM relation-list
[WHERE conditions];
```

SELECT Clause

Select Distinct

```
SELECT DISTINCT country_iso AS code
FROM cities;
```

String concatenation

```
SELECT
    name,
    '$' || value AS gdp
FROM countries ;
-- 'Denmark', '$500'
```

WHERE Clause

Value range conditions

```
SELECT * FROM countries
WHERE (population BETWEEN 500 AND 600);
```

Pattern Matching

- `_` matches any single character
- `%` matches any sequence of 0 or more characters

```
SELECT name FROM cities
WHERE name LIKE 'Si%re';
-- Singapore, Sire, Sierra Madre
```

Set operations

Let Q_1, Q_2 be SQL queries that yield union-compatible tables

- $Q_1 \cup Q_2$: UNION
- $Q_1 \cap Q_2$: INTERSECT
- $Q_1 \setminus Q_2$: EXCEPT

```
(SELECT value FROM R)
UNION
(SELECT value from S);
```

- UNION, INTERSECT, EXCEPT
 - ▷ No duplicate tuples
- UNION ALL, INTERSECT ALL, EXCEPT ALL
 - ▷ Include duplicate tuples

JOIN Operations

Cross Product

- Returns every possible combination of $A \times B$ with cardinality $|A| \times |B|$

```
SELECT * FROM cities CROSS JOIN countries;
SELECT * FROM cities, countries; --alt
```

Conditional Cross Product

```
SELECT c.name, n.name
FROM cities c, countries n
WHERE c.country_iso = n.iso;
```

```
SELECT c.name, n.name
FROM cities c INNER JOIN countries n
ON c.country_iso = n.iso
```

```
SELECT c.name, n.name
FROM cities c JOIN countries n
ON c.country_iso = n.iso
```

Multiple Joins

Joining n tables requires $n - 1$ join conditions.

```
SELECT
    c1.name, c1.population,
    c2.name, c2.population
FROM
    countries c1,
    borders b,
    countries c2
WHERE
    c1.iso = b.country1_iso AND --join cond 1
    c2.iso = b.country2_iso AND --join cond 2
    c1.population < c2.population
```

Natural Joins

Join conditions can be implied by attribute names.

```
SELECT DISTINCT (name)
FROM
    (SELECT name FROM cities) t1
    NATURAL JOIN
    (SELECT name FROM countries) t2;
```

Outer Joins

- LEFT OUTER JOIN : Inner join + remaining left tuples
- FULL OUTER JOIN : Inner join + all remaining tuples
- RIGHT OUTER JOIN : Inner join + remaining right tuples

```
SELECT n.name
FROM countries n
LEFT JOIN cities c ON n.iso = c.country_iso
WHERE c.country_iso IS NULL;
-- keep dangling tuples
```

Nested Queries

IN Subqueries

Subquery must return exactly 1 column. True if expr matches with any subquery row, FALSE otherwise.

```
SELECT name FROM countries
WHERE name IN (SELECT name FROM cities);
```

```
SELECT * FROM countries
WHERE
    continent IN ('Asia', 'Europe') AND
    population BETWEEN 5000 AND 6000;
```

ANY / ALL Subqueries

- Subquery must return exactly 1 column. We then compare expr to each subquery row using operator op
- ANY returns TRUE if comparison is true for at least 1 subquery row
- ALL return TRUE if comparison is true for all subquery rows

```
SELECT name, population FROM countries
WHERE population < ALL (
    SELECT population FROM cities
    WHERE name = 'London'
);
```

Correlated Subquery

Subquery uses value from outer query.

```
SELECT name, continent, gdp
FROM countries c1
WHERE gdp >= ALL (
    SELECT gdp
    FROM countries c2
    WHERE c2.continent = c1.continent
    AND c2.gdp IS NOT NULL -- important check
);
```

Exists Subquery

- Returns TRUE if the subquery returns at least one tuple

```
SELECT n.name FROM countries n
WHERE EXISTS (
    SELECT c.name FROM cities c
    WHERE c.name = n.name);
```

* NOT EXISTS subqueries are generally always correlated

Scalar Subqueries

- Subquery returns a single value (1 row 1 col)
- This may then be used as a expression in queries

```
SELECT
    name AS city
    (SELECT name AS country
    FROM countries n
    WHERE n.iso = c.country_iso)
    FROM cities c;
```

Row Constructors

- We have a row, a operator op and a keyword ANY / ALL, and then we can perform row-wise comparison with a compatible row

```
SELECT name, population, gdp FROM countries
WHERE ROW(population, gdp) > ANY (
    SELECT population, gdp
    FROM countries
    WHERE name IN ('France', 'Germany')
);
```

Sorting

- Order priority is left to right (stable)

```
SELECT name
FROM names
ORDER BY names.left ASC, names.right DESC;
```

Limit and Offset

- LIMIT k : return first k tuples
- OFFSET i : i is the first tuple under consideration

```
SELECT name FROM countries
WHERE gdp IS NOT NULL
ORDER BY gdp DESC
OFFSET 5
LIMIT 5;
-- take 2nd top 5 highest gdps
```

Aggregation

- Computes a single value of a column from a set of tuples

Minimum non-NULL

```
SELECT MIN(A) FROM R;
```

Maximum non-NULL

```
SELECT MAX(A) FROM R;
```

Mean of non-NULL

```
SELECT AVG(A) FROM R;
```

Sum of non-NULL

```
SELECT SUM(A) FROM R;
```

Count of non-NULL

```
SELECT COUNT(A) FROM R;
```

Count of all rows

```
SELECT COUNT(*) FROM R;
```

Mean of distinct non-NULL

```
SELECT AVG(DISTINCT A) FROM R;
```

Sum of distinct non-NULL

```
SELECT SUM(DISTINCT A) FROM R;
```

Count of distinct non-NULL

```
SELECT COUNT(DISTINCT A) FROM R;
```

Grouping

- Applied with aggregation to partition relation into groups based on specific attributes

```
SELECT continent
    MIN(population) AS lowest,
    MAX(population) AS highest,
    SUM(population) AS overall
    FROM countries
    GROUP BY continent;
```

Conditions over Groups

- HAVING is used with GROUP BY

```
SELECT
    from_code,
    to_code
    COUNT(*) AS num_airlines
    FROM routes
    GROUP BY from_code, to_code
    HAVING COUNT(*) > 12;
```

Coalesce

- Returns the first non-NULL value in its argument list

```
SELECT
    name,
    COALESCE(end_date, CURRENT_DATE) AS
    latest_day
    FROM Employees;
```

Case

```
SELECT
    name,
    CASE
        WHEN score >= 90 THEN 'A'
        WHEN score >= 80 THEN 'B'
        ELSE 'C'
    END AS grade
    FROM student;
```

```
SELECT
    name,
    CASE role
        WHEN 'admin' THEN 1
        WHEN 'user' THEN 2
        ELSE 3
    END AS role_id
    FROM accounts;
```

Conditionals for Null

- `NULLIF(v1, v2)` returns `NULL` if `v1 == v2` else `v1`

```
SELECT
  MIN(NULLIF(gini, 0)) AS min_gini,
  AVG(NULLIF(gini, 0)) AS avg_gini
FROM countries;
-- avoid 0s
```

Structuring Queries

Common Table Expressions

- Temporary named query

```
WITH
  HighEarners AS (
    SELECT name, department, salary
    FROM employees
    WHERE salary > 5000
  )
  SELECT
    department,
    COUNT(*) AS num_high_earners
  FROM HighEarners
  GROUP BY department;
```

Views

- Permanently named query

```
CREATE VIEW HighEarners AS
SELECT name, department, salary
FROM employees
WHERE salary > 5000;

SELECT
  department,
  COUNT(*) AS num_high_earners
FROM HighEarners
GROUP BY department;
```

Recursive queries

Recursive Queries with Common Table Expressions

```
WITH RECURSIVE cte AS (
  Q1 --base case, cannot reference cte
  UNION
  Q2(cte) -- recursive term w cte
)
SELECT * FROM cte;
```

```
WITH RECURSIVE flight AS (
  -- base case
  SELECT from_code, to_code, 0 AS stops
  FROM connections
  WHERE from_code = 'SIN'
  UNION ALL
  -- recursive case
  SELECT c.from_code, c.to_code, p.stops +
  1
  FROM flight p, connections c
  WHERE p.to_code = c.from_code
  AND p.stops < 2
)
SELECT DISTINCT to_code, stops
FROM flight
ORDER BY stops ASC;
```

Stored Procedures and Functions

Function Declaration

```
CREATE OR REPLACE FUNCTION func(dob DATE)
RETURNS INTEGER AS $$
DECLARE
  days INTEGER;
BEGIN
  --- logic here
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Function Invocation

With `SELECT`

```
SELECT func('2020-02-02');
```

As part of query

```
SELECT c.id, func(c.dob) AS age
FROM customers c
ORDER BY age;
```

Assignment

Storing value from Query Result

```
DECLARE
  v_price NUMERIC;
  v_pages INT;
BEGIN
  SELECT price, pages
  INTO v_price, v_pages
  FROM book
  WHERE isbn13 = '12345678901234';
END;
```

Storing tuple from Query Result

```
DECLARE
  rec RECORD;
BEGIN
  SELECT *
  INTO rec
  FROM book
  WHERE isbn13 = '12345678901234';
END;
```

* `SELECT ... INTO` sets the special variable `FOUND` to true if at least one row was retrieved, false otherwise.

Assignment with `SELECT`

```
years := EXTRACT(year FROM CURRENT_DATE) -
EXTRACT(year FROM dob);
```

Control Flow

```
IF condition1 THEN
  -- branch 1
ELSIF condition2 THEN
  -- optional
  -- branch 2
ELSE
  -- fallback
END IF;
```

Loops

While Loops

```
-- BEGIN
days := 0;
WHILE dob < CURRENT_DATE LOOP
  days := days + 1
  dob := dob + 1;
END LOOP;
RETURN days;
-- END;
```

Return from Existing Schema

Returning a Single Tuple from Existing Table

- Specify the table name as the return type

```
CREATE OR REPLACE FUNCTION most_expensive()
RETURNS games AS $$
SELECT *
FROM games g
ORDER BY g.price DESC
LIMIT 1;
$$ LANGUAGE sql;
```

Returning a table from Existing table

- Using `SETOF`

```
CREATE OR REPLACE FUNCTION all_most_ex()
RETURNS SETOF games AS $$%
SELECT *
FROM games g
GROUP BY g.name, g.version
HAVING g.price >=
ALL(SELECT gl.price FROM games gl);
$$ LANGUAGE sql;
```

Returning Arbitrary Tuple

Using `OUT` parameters

- If a function describes `OUT` parameters, psql automatically treats them as the fields of a single returned row

```
CREATE OR REPLACE FUNCTION demo
(a int, OUT x int, OUT y text) AS $$%
BEGIN
  x := a + 1;
  y := 'hello';
END;
$$ LANGUAGE plpgsql;

SELECT * FROM demo(5); -- (x: 6, y: hello)
```

Using `RECORD` as Return type

```
CREATE OR REPLACE FUNCTION demo_rec(a int)
RETURNS RECORD AS $$
DECLARE
  result RECORD;
BEGIN
  SELECT a + 1 AS x, 'hello' AS y
  INTO result;
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Using `INOUT` Parameters

```
CREATE OR REPLACE FUNCTION foo(INOUT value
int)
AS $$%
BEGIN
  value := value + 1;
END;
$$ LANGUAGE plpgsql;

SELECT foo(10); -- (11)
```

Returning Arbitrary Table

SETOF RECORD

```
CREATE OR REPLACE FUNCTION all_most_download
(OUT name VARCHAR(32), OUT version CHAR(3))
RETURNS SETOF RECORD AS $%
SELECT g.name, g.version
FROM games g, downloads d
WHERE
  g.name = d.name AND
  g.version = d.version
GROUP BY g.name, g.version
HAVING COUNT(*) >= ...
$$ LANGUAGE sql;
```

TABLE(...)

```
CREATE OR REPLACE FUNCTION all_most_download
RETURNS TABLE(name VARCHAR(32), version
CHAR(3))
AS $$ ...
```

Manipulating Tables with Procedures

Inserting with `plpgsql`

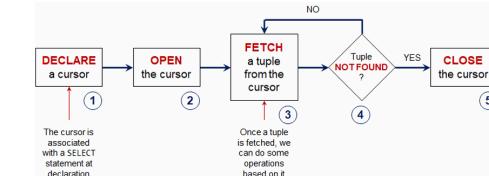
```
CREATE OR REPLACE PROCEDURE download_game
(cid VARCHAR(16), gname VARCHAR(32), gver
CHAR(3)) AS $$%
BEGIN
  INSERT INTO downloads VALUES
  (cid, game, gver)
END;
$$ LANGUAGE plpgsql;
```

Deleting with `plpgsql`

```
-- BEGIN
DELETE FROM downloads d
WHERE d.customerid IN (
  SELECT c.customerid FROM customers c...
);
-- END;
```

Cursor

- Behaves like an iterator over a SQL query



Example: Largest price jump

```
CREATE OR REPLACE FUNCTION max_increase(gname
VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE
    cur CURSOR (vname VARCHAR(32)) FOR
        SELECT g.price
        FROM games g
        WHERE g.name = vname
        ORDER BY g.version ASC;

    res NUMERIC; -- stores the maximum
increase
    prev NUMERIC; -- previous row's price
    curr NUMERIC; -- current row's price
BEGIN
    OPEN cur(vname := gname);
    res := 0;
    prev := 0;
    LOOP
        -- Fetch next price into curr
        FETCH cur INTO curr;
        EXIT WHEN NOT FOUND;
        IF (curr - prev) >= res THEN
            res := (curr - prev);
        END IF;
        prev := curr;
    END LOOP;
    CLOSE cur;
    RETURN res;
END;
$$ LANGUAGE plpgsql;
```

Fetching a tuple from Cursor

```
DECLARE
    curs CURSOR FOR (...);
    rec RECORD;
BEGIN
    OPEN curs;
    LOOP
        FETCH NEXT FROM curs INTO rec;
        EXIT WHEN NOT FOUND;
        x = rec.email; -- do something
    END LOOP;
    CLOSE curs;
    RETURN;
END;
```

Returning a table from Cursor

```
CREATE OR REPLACE FUNCTION return_table()
RETURNS TABLE (id int, diff BIGINT) AS $$
DECLARE ...
BEGIN
    ...
LOOP
    id := temp.id;
    diff := temp.diff;
    RETURN NEXT; -- similar to yield
END LOOP;
...
END;
```

Raising Exceptions

Level	Note
RAISE DEBUG	
RAISE LOG	Generate messages of different priority
RAISE INFO	
RAISE NOTICE	Whether messages are displayed to the client depends on postgresql configuration
RAISE WARNING	
RAISE EXCEPTION	Raises an error. Typically aborts the current transaction.

Basic

```
RAISE EXCEPTION 'Something went wrong.';
-- ERROR: Something went wrong
```

With variables

```
RAISE NOTICE
    'book: %, email: %', book, email;
```

Triggers

- BEFORE :
 - ▶ Runs before the row operation
 - ▶ Can modify the row being inserted/updated by returning NEW
 - ▶ Can prevent an operation by returning NULL
- AFTER :
 - ▶ Runs after the row operation has been performed
 - ▶ Cannot change the row
 - ▶ Can perform actions that depend on the completed change

```
CREATE OR REPLACE FUNCTION trigger_func()
RETURNS TRIGGER AS $$
BEGIN
    -- Example side-effect
    IF EXISTS (...) THEN
        RAISE EXCEPTION 'ERROR';
        -- Can stop execution with EXCEPTION
        -- causes a rollback
    END IF;
    -- Must return either NEW or OLD
    IF TG_OP = 'INSERT' OR
        TG_OP = 'UPDATE' THEN
        RETURN NEW;
    ELSE
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE CONSTRAINT TRIGGER my_trigger
AFTER INSERT OR UPDATE ON my_table
DEFERRABLE INITIALLY DEFERRED -- optional
FOR EACH ROW
EXECUTE FUNCTION trigger_func();
```

Logging Example

```
CREATE OR REPLACE FUNCTION log_ops()
RETURNS trigger AS $$
BEGIN
    INSERT INTO downloads_log VALUES (
        NEW.customerid,
        NEW.name,
        TG_OP,
        CURRENT_DATE
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER op_log
AFTER INSERT OR UPDATE OR DELETE ON downloads
FOR EACH ROW
EXECUTE FUNCTION log_ops();
```

Triggers on Views

- We can use INSTEAD OF to perform operations on a view.

```
CREATE TABLE items (
    id int PRIMARY KEY,
    qty int
);
CREATE VIEW item_view AS
SELECT id, qty
FROM items;
```

```
CREATE OR REPLACE FUNCTION
update_item_view_fn()
RETURNS trigger AS $$
BEGIN
    UPDATE items
    SET qty = NEW.qty
    WHERE id = NEW.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER update_item_view
INSTEAD OF UPDATE ON item_view
FOR EACH ROW
EXECUTE FUNCTION update_item_view_fn();
-- Perform updates on views:
-- UPDATE item_view SET qty = 20 WHERE id = 1
```

Effects of Return Value

- NEW contains the modified row after trigger event
 - ▶ INSERT , UPDATE
- OLD contains the modified row before trigger event
 - ▶ DELETE , UPDATE

	NULL Tuple	Non-NULL Tuple t
BEFORE INSERT	No tuple inserted	Tuple t will be inserted
BEFORE UPDATE	No tuple updated	Tuple t will be the updated tuple
BEFORE DELETE	No tuple deleted	Deletion proceed as normal
AFTER ...		No effect

Conditional Trigger

```
CREATE TRIGGER foo
BEFORE INSERT ON score
FOR EACH ROW
WHEN (NEW.name = 'bar') -- Condition
EXECUTE FUNCTION bar();
```

WHEN conditions:

- No SELECT
- No OLD for INSERT
- No NEW for DELETE
- No WHEN for INSTEAD OF

Trigger Granularity and Order

FOR EACH ROW

- Trigger executes once per row that is modified
- has NEW and OLD

FOR EACH STATEMENT

- Trigger executes once per statement, even if multiple rows are edited
- no NEW and OLD

Activation Order

1. BEFORE Statement level
2. BEFORE Row-level
3. AFTER Row-level
4. AFTER Statement level

Within each category, triggers are activated in alphabetical order

Removing triggers

```
DROP TRIGGER trigger ON my_table;
DROP FUNCTION trigger_func();
```

Constraint trigger

- Used for enforcing constraints that need to be checked at the end of a transaction

```
CREATE CONSTRAINT TRIGGER my_trg
AFTER INSERT ON my_table
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION foo();
```